Die .NET Common Language Runtime : Überblick und technischer Einstieg

Autor(en): **Willers, Michael**

Objekttyp: **Article**

Zeitschrift: **Bulletin.ch : Fachzeitschrift und Verbandsinformationen von Electrosuisse, VSE = revue spécialisée et informations des associations Electrosuisse, AES**

Band (Jahr): **94 (2003)**

Heft 1

PDF erstellt am: **29.05.2024**

Persistenter Link: https://doi.org/10.5169/seals-857511

Nutzungsbedingungen

Die ETH-Bibliothek ist Anbieterin der digitalisierten Zeitschriften. Sie besitzt keine Urheberrechte an den Inhalten der Zeitschriften. Die Rechte liegen in der Regel bei den Herausgebern.
Die auf der Plattform e-periodica veröffentlichten Dokumente stehen für nicht-kommerzielle Zwecke in Lehre und Forschung sowie für die private Nutzung frei zur Verfügung. Einzelne Dateien oder Ausdrucke aus diesem Angebot können zusammen mit diesen Nutzungsbedingungen und den korrekten Herkunftsbezeichnungen weitergegeben werden.
Das Veröffentlichen von Bildern in Print- und Online-Publikationen ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Die systematische Speicherung von Teilen des elektronischen Angebots auf anderen Servern bedarf ebenfalls des schriftlichen Einverständnisses der Rechteinhaber.

Haftungsausschluss

Alle Angaben erfolgen ohne Gewähr für Vollständigkeit oder Richtigkeit. Es wird keine Haftung übernommen für Schäden durch die Verwendung von Informationen aus diesem Online-Angebot oder durch das Fehlen von Informationen. Dies gilt auch für Inhalte Dritter, die über dieses Angebot zugänglich sind.

Ein Dienst der *ETH-Bibliothek*
ETH Zürich, Rämistrasse 101, 8092 Zürich, Schweiz, www.library.ethz.ch

http://www.e-periodica.ch

Die .NET Common Language Runtime – Überblick und technischer Einstieg

Das .NET Framework ist der für Entwickler wichtigste Bestandteil der neuen .NET-Plattform von Microsoft. Herzstück des Frameworks ist die Common Language Runtime. Grund genug also, ein wenig hinter die Kulissen zu schauen und die Konzepte dieser Laufzeitumgebung näher zu beleuchten. Dieser Artikel gibt einen Überblick und liefert einen Einstieg in die Thematik.

Die meist gestellte Frage bei der Einführung neuer Technologien ist die nach dem Warum. Warum also eine gemeinsame Laufzeitumgebung? Blicken wir dazu ein wenig zurück. Das Anfang der 90er-Jahre von Microsoft eingeführte Component Object Model (COM) sollte dazu dienen, die Kommunikation und Integration zwischen Softwarekomponenten zu vereinheitlichen. Dieses Modell hat sich am Markt durchgesetzt. Das belegen die zahlreichen Lösungen, die heute verfügbar sind – von einfachen visuellen Komponenten (Controls) bis hin zur kompletten Anbindung an SAP-Systeme (SAP/R3).

Michael Willers

Die Idee dahinter: Die Integration sollte binär und somit unabhängig von Programmiersprachen erfolgen. Um diese Integration zu ermöglichen, bringt COM ein eigenes Typsystem mit. Kurz gesagt: Jede Sprache muss neben dem eigenen Typsystem zusätzlich das Typsystem von COM implementieren, um interoperabel zu sein. Diese Vorgehensweise bringt im Wesentlichen drei Probleme mit sich:

- Die Sprache benötigt einen Layer, der das Typsystem von COM implementiert.
- Der Entwickler muss über diesen Layer die Konvertierungen von Sprachtypen in COM-Typen und umgekehrt von Hand programmieren.
- Der Entwickler muss neben der Konvertierung zusätzlichen Infrastrukturcode für den Layer programmieren, um den Aufrufkonventionen von COM zu genügen.

Diese zusätzliche Programmierung macht den Code komplex und extrem fehleranfällig. Wer als C++-Programmierer häufig mit Feldern oder Zeichenketten (COM-Typ SAFEARRAY bzw. BSTR) arbeitet, der weiss, wovon hier die Rede ist.

Im Laufe der Jahre ist dieses Modell weiterentwickelt und um Dienste für verteilte Anwendungen sowie die Möglichkeit für Fernaufrufe (Microsoft Transaction Server, MTS bzw. Distributed COM, DCOM) erweitert worden. Diese Weiterentwicklungen sind mit Windows 2000 zu einem einheitlichen Modell zusammengeflossen, das unter der Bezeichnung COM+ weitläufig bekannt ist. Aber auch COM+ konnte die Basisprobleme nicht lösen: Jede Sprache benötigt weiterhin einen Layer, der das Typsystem von COM implementiert, und der Entwickler muss durch zusätzliche Programmierung seine Spache an COM anpassen .

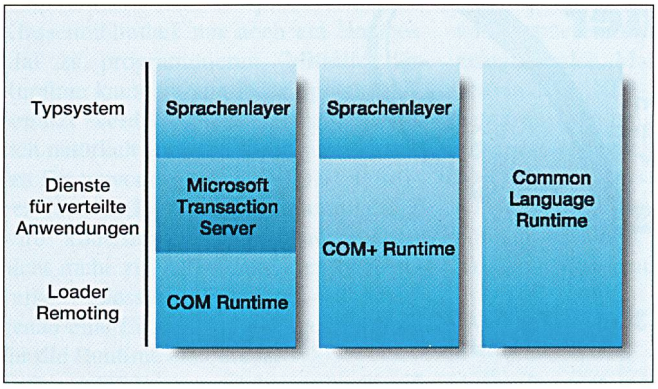

Bild 1 Die Common Language Runtime bietet ein einheitliches Integrationsmodell

Hier setzt die Common Language Runtime (CLR) an und bietet ein einheitliches Integrationsmodell (Bild 1). Die eingangs gestellte Frage nach dem Warum kann also wie folgt beantwortet werden: Die Common Language Runtime bietet ein einheitliches Integrationsmodell, und dieses Modell sorgt dafür, dass die Anwendungsentwicklung konsistenter und einfacher wird.

Codemanager

Im .NET Framework spielt der Begriff *Managed* eine zentrale Rolle. So wird Code, der unter der Regie der Runtime ausgeführt wird, mit *Managed Code* bezeichnet. Das bedeutet, dass Aktionen wie das Anlegen eines Objekts oder der Aufruf einer Methode nicht direkt erfolgen, sondern an die Runtime delegiert werden. Diese kann dann zusätzliche Dienste wie beispielsweise Versionsüberprüfungen ausführen.

Aus diesem Grund erzeugen die Compiler des Frameworks keinen native Code mehr. Vielmehr wird aus dem Quelltext eine prozessorunabhängige Zwischensprache erzeugt (Bild 2), die dann unter Aufsicht der Runtime bei Bedarf zu native Code kompiliert und ausgeführt wird (Just in time Compiler, JIT). Diese Zwischensprache wird mit Common Intermediate Language, CIL, kurz IL, bezeichnet. IL-Code wird vor der Ausführung grundsätzlich in echten Maschinencode übersetzt. Somit ist gewährleistet, dass immer die schnellstmöglichste Ausführungsgeschwindigkeit gegeben ist. Zudem erlaubt dieses Verfahren eine Entkopplung der Runtime von der zugrunde liegenden Hardware. Ein Vorläufer ist bereits unter Windows CE im Einsatz. Dort erzeugen die Compiler der Entwicklungumgebung auch eine Zwischensprache; erst beim Download auf das CE-Gerät wird die Anwendung in native Code übersetzt und damit hardwareabhängig.

Einzige Ausnahme: Der Visual-C++-Compiler. Er auch kann weiterhin native Code erzeugen. Hier ist der Grund allerdings einleuchtend. Nur ein Bruchteil aller Entwickler kommt mit der rauhen Wirklichkeit der Programmierung in Berührung: Die Kernel- und Treiberent-

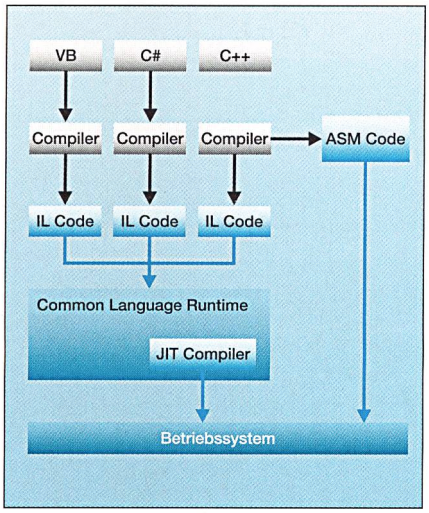

Bild 2 Sprachintegration erfolgt zukünftig auf Codeebene

Visual C++ kann allerdings weiterhin native Code erzeugen, damit es auch zukünftig noch performante Treibersoftware gibt.

```
' ------------------------------------
' Die Basisklasse in VB.NET formuliert
' Datei BASICDOG.VB
' ------------------------------------
Option Strict Off
Namespace DogClone
Public Class BasicDog
  Public Sub Bark(ByVal strName as System.String)
      System.Console.WriteLine("Hello, {0}!",strName)
  End Sub
  Public Sub Ouch
      Dim x as integer = 0
      Dim y as integer = 42 / x
      System.Console.WriteLine("Ouch")
  End Sub
End Class
End Namespace

// ----------------------------------------
// Eine abgeleitete Klasse in C# formuliert
// Datei EXTDOG.CS
// ----------------------------------------
namespace DogClone {
  public class ExtendedDog : BasicDog {
      public void Growl(string strName) {
          System.Console.WriteLine("Hello, {0}. I will bite you!",strName);
      }
  }
}

// -------------------------------
// Und noch ein Testprogramm in C#
// Datei DOGDEMO.CS
// -------------------------------
using DogClone;

namespace DogApp {
  class EntryPoint {
      static void Main() {
          try {
              ExtendedDog myDog = new ExtendedDog();
              myDog.Bark("Michael");
              myDog.Ouch();
              myDog.Growl("Simon");
          }
          catch (System.Exception e) {
              System.Console.WriteLine(e);
          }
          System.Console.ReadLine();
      }
  }
}
```

Listing 1 Vererbung und einheitliche Fehlerbehandlung über Sprachgrenzen hinweg

wickler. Für diese Einsatzgebiete ist eine Runtime denkbar ungeeignet, da in der Regel eine Plattformabhängigkeit vorliegt und maximale Performance zwingend erforderlich ist.

Wenn die Methode 1 (IL) der Klasse A aufgerufen wird, sucht diese in der Klasse B die Methode 1 (ASM). Falls diese vorhanden ist, wird sie aktiviert, wenn nicht, wird die IL-Methode vom JIT-Compilers in eine ASM-Methode übersetzt und dann ausgeführt. Die Common Intermediate Language ist komplett offen gelegt; sie wurde im Dezember 2001 von der europäischen Standardisierungsbehörde ECMA als Standard verabschiedet. Weitere Informationen findet man unter http://msdn.microsoft.com/net/ecma.

Somit kann jeder Compiler, der IL-Code erzeugt, diesen unter Aufsicht der Runtime ausführen lassen. Oder anders gesagt: Dreh- und Angelpunkt der Runtime ist die Integration auf Codeebene. Ob man nun Cobol, Pascal, C# oder Visual Basic benutzt, ist egal, solange der Compiler IL-Code erzeugt. Man kann nun beispielsweise eine Klasse in einer Sprache erstellen und mittels einer anderen Sprache eine weitere Klasse davon ableiten (Bild 3). Die Bedeutung, welche Sprache man zur Entwicklung von Anwendungen benutzt, rückt damit in den Hintergrund. Man arbeitet mit der Sprache, die einem am besten liegt (Listing 1).

Allen .NET-Compilern ist übrigens gemeinsam, dass sie die umfangreiche Klassenbibliothek der Runtime benutzen. Diese vereinheitlicht die bisherigen Programmierschnittstellen zu einem gemeinsamen Modell und bietet Klassen für nahezu jede Lebenslage – von Basisklassen für die Bearbeitung von Zeichenketten bis hin zu Klassen für die Thread-Programmierung. Zugriffe auf das Betriebssystem und das Win32-API erfolgen also nicht mehr direkt, sondern werden über Klassen abstrahiert.

Ein weiterer Bereich der Bibliothek befasst sich mit Fehlerbehandlung – sie erfolgt über alle Sprachen einheitlich in Form von Exceptions.

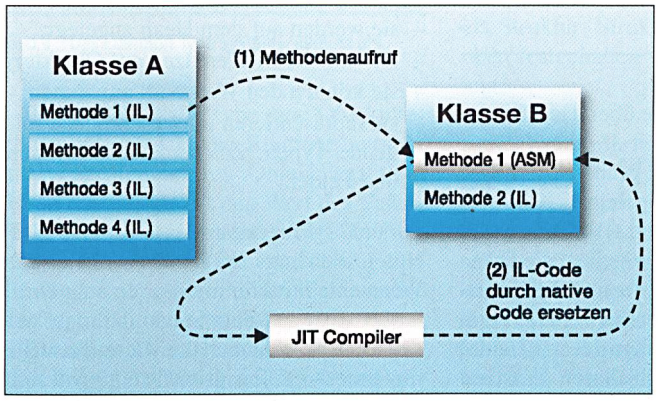

Bild 3 IL-Code wird durch den JIT-Compiler der Common Language Runtime bei Bedarf in native Code übersetzt und ausgeführt.

Der Compiler wird zur Laufzeit erst dann benötigt, wenn der Methodenaufruf erfolgt – und nicht vorher. IL: Intermediate Language; JIT: Just In Time; ASM: Assembler Language

Das Common Type System

Die Idee der Integration auf Codeebene geht allerdings noch einen Schritt weiter. Die Common Language Runtime stellt allen .NET-Sprachen ein umfassendes Typsystem zur Verfügung (Bild 4). Kurz gesagt: Das Typsystem wandert

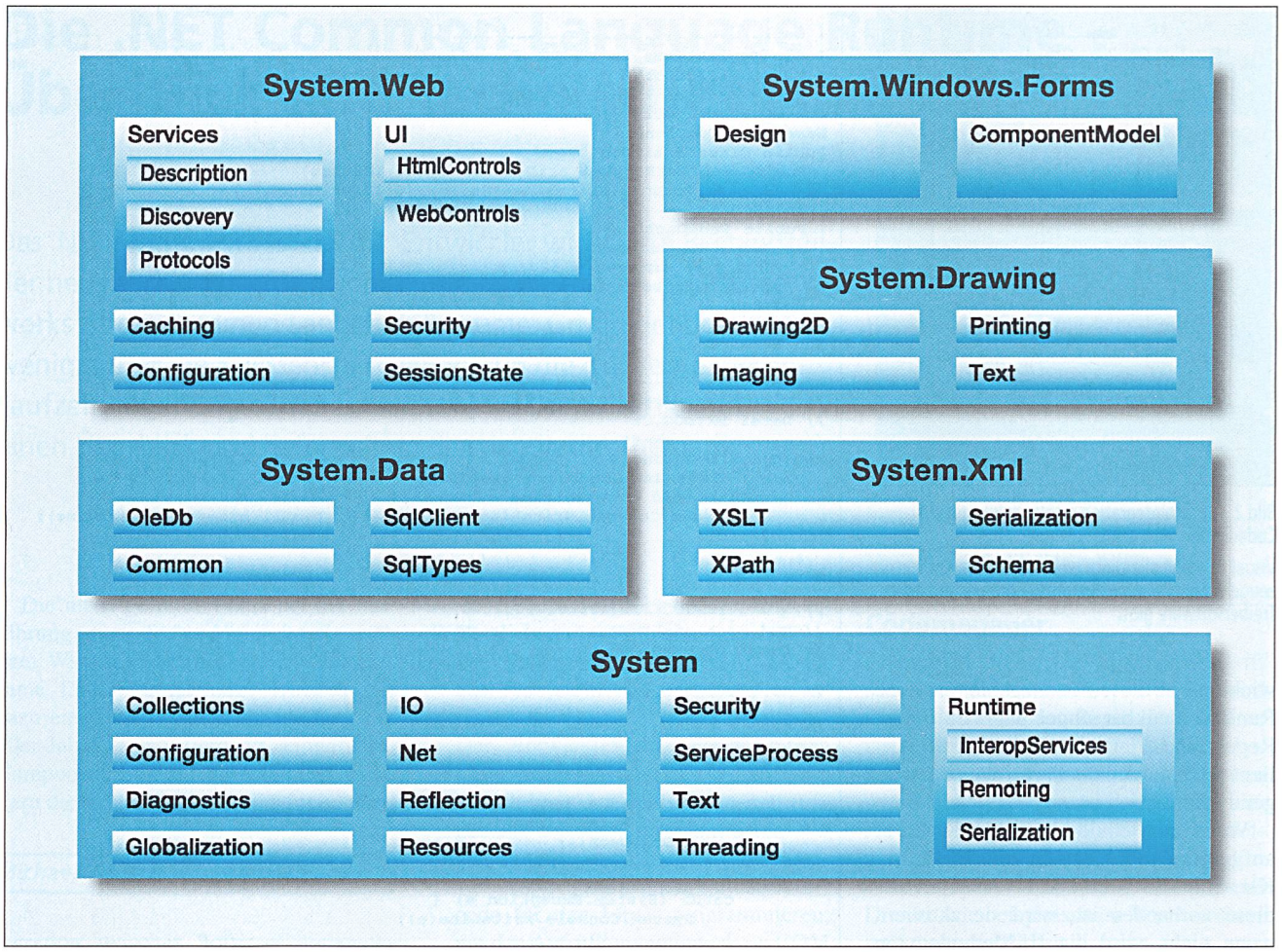

Bild 4 Die Klassenbibliothek im .NET Framework – ein einheitliches Programmiermodell

vom Compiler in die Runtime. Es ist nicht mehr Bestandteil einer Sprache. Vielmehr setzen alle Sprachen auf dem Common Type System (CTS) der Runtime auf. Das bedeutet: Typen werden eindeutig, da es nicht mehr verschiedene Repräsentationen ein und desselben Typs gibt – so ist beispielsweise eine Zeichenkette unter Visual Basic.NET identisch mit einer Zeichenkette unter C#.

Typkonvertierungen und Anpassungen an COM-Aufrufkonventionen (Bild 5a) sind somit nicht mehr erforderlich, wenn Komponenten unterschiedlicher Sprachen miteinander kommunizieren. Sprachen sind per Definition interoperabel (Bild 5b), da sie das gleiche Typsystem benutzen.

Für Anwendungsentwickler und Compilerbauer wird es einfacher. Der Anwendungsentwickler wird von fehleranfälligem Konvertierungscode entlastet und der Compilerbauer muss weder Typsystem noch eine Klassenbibliothek implementieren, da beides fester Bestandteil der CLR ist. Andererseits muss Letzterer dafür sorgen, das bei der Portierung einer bereits bestehenden Sprache auf die CLR ein Mapping von Sprachtypen auf die Typen des Common Type Systems erfolgt. Aus der Sicht des Anwendungsentwicklers ist dies transparent, er benutzt wie gewohnt seine Sprache. Das Mapping erfolgt «behind the scenes»; beim Übersetzen durch den Compiler wird der entsprechende IL-Code erzeugt (Visual C++.NET ist ein Beispiel für eine solche Portierung).

Alles ist Objekt?

Sämtliche Typen, die über das CTS definiert sind, werden mit *Managed Types* bezeichnet. Sie werden grundsätzlich von Typ *System.Object* abgeleitet. Kurz gesagt: Alles ist ein Objekt (Bild 6).

Dabei stellt sich automatisch die Frage nach der Performance, denn Objekte werden normalerweise stets auf einem eigenen Speicherbereich, dem Heap (Halde) abgelegt. Betrachen wir diesen Umstand ein wenig genauer. Grundsätzlich wird zwischen zwei Arten von Typen unterschieden: *ValueType* und *ReferenceType*.

ValueTypes zeichnen sich durch folgende Eigenschaften aus:

- sie werden auf dem Stack angelegt;
- sie enthalten Daten;
- sie können nicht dem Wert Null annehmen;
- sie repräsentieren im Wesentlichen folgende Typen: Primitive Datentypen wie int, Aufzählungen und Strukturen.

Im Gegensatz dazu gilt für *ReferenceTypes*:

- sie werden auf dem Heap angelegt;
- sie enthalten Referenzen auf Objekte;
- sie können den Wert Null annehmen;
- sie repräsentieren im Wesentlichen folgende Typen: Zeichenketten, Klassen und Felder.

Für C++-Programmierer ist das ein alter Hut. Allerdings mit einer Ausnahme: Wenn eine Struktur mit *new* erzeugt wird, landet diese auf dem Stack und nicht auf dem Heap. Man behalte diese Tatsache im Hinterkopf. Primitive Datentypen sind also als ValueTypes definiert und werden somit auf dem Stack abgelegt. Wie aber kann dann ein ValueType eine Objektmethode aufrufen, wenn doch alle Typen von *System.Object* abgeleitet werden?

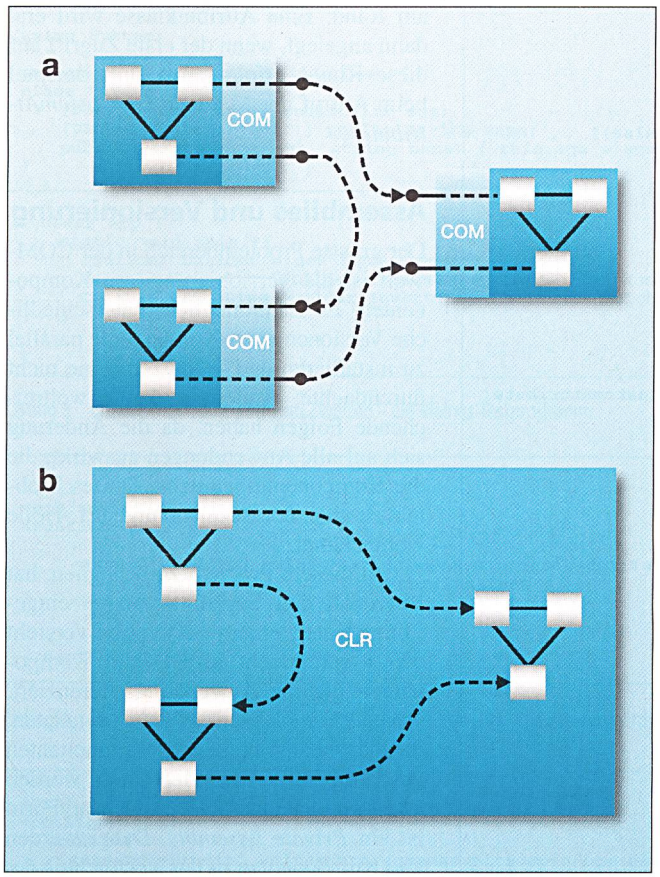

Bild 5 Programmieren unter COM und Common Language Runtime

a: Unter COM wird jeder Sprache ein eigener Layer übergestülpt, um interoperabel zu sein. Der Entwickler muss Typkonvertierungen erledigen und sich an die Aufrufkonventionen von COM halten. Die Folge: Komplexer Code und Fehler sind vorprogrammiert.
b: Die Common Language Runtime sorgt dafür, dass Sprachen per Definitionem interoperabel sind und stellt eine entsprechende Infrastruktur bereit. Typkonvertierungen und das Einhalten von COM-Aufrufkonventionen sind nicht mehr notwendig.

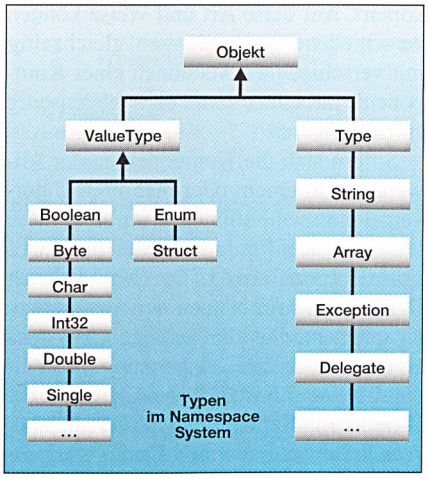

Bild 6 Das Common Type System – alles ist ein Objekt

Sobald ein ValueType eine Objektmethode aufruft, legt die Runtime automatisch ein temporäres Objekt auf dem Heap an und kopiert den Wert des ValueTypes dort hinein. Dann erfolgt der Methodenaufruf. Nach der Ausführung des Methodenaufrufs wird das Objekt wieder entfernt und man arbeitet mit dem ValueType weiter. Diese Techniken wurden mit den Begriffen *Boxing* und *Unboxing* getauft. Mit Boxing wird das Konvertieren eines ReferenceType in einen ValueType bezeichnet, mit Unboxing der umgekehrte Vorgang. Man kann Boxing und Unboxing allerdings auch explizit einsetzen, um das Arbeiten mit temporären Objekten zu vermeiden (Bild 7).

Im Hinblick auf die Performance ist diese Vorgehensweise ein guter Kompromiss. Es ist nur dann ein echtes Objekt auf dem Heap vorhanden, wenn es wirklich gebraucht wird. Für den Entwickler ist dies vollkommen transparent. Aus seiner Sicht sind alle Typen Objekte und die Runtime erledigt den Rest hinter den Kulissen.

Diese Tatsache ist dann von Bedeutung, wenn man eigene Typen implementiert, um auf einfache Art und Weise «leichtgewichtige» Objekte zu erzeugen. Man definiert den eigenen Typ einfach als Struktur; Strukturen werden ja – wie oben festgehalten – auf dem Stack abgelegt.

Metadaten und Reflection

Komponenten, die mit der Runtime programmiert wurden, beschreiben sich selbst. Entsprechende Metadaten werden beim Übersetzen durch den Compiler in die Komponente geschrieben. Diese Metadaten können zur Laufzeit ausgelesen und verarbeitet werden. Man bezeichnet diesen Vorgang mit Reflection – ein für JAVA-Anhänger bereits bekanntes Verfahren. Der Vorteil: Der Installationsvorgang beschränkt sich auf einfaches Kopieren; zusätzliche Dateien zur Beschreibung – wie Headerdateien oder Typenbibliotheken in der COM-Welt – sind nicht mehr erforderlich.

Die Metadaten enthalten die Beschreibungen sämtlicher Typen, die in einer Komponente definiert sind (Bild 8). Dazu zählen Schnittstellen, Klassen und deren Membervariablen usw. Die Membervariablen werden übrigens *Felder* genannt. Wie aber lassen sich die Metadaten auslesen? Sobald die Common Language Runtime einen Typ erzeugt (z.B. eine Klasse), wird gleichzeitig ein Objekt vom Typ *System.Type* angelegt und der soeben erzeugten Typinstanz zugeordnet. Über die Methoden dieses Typobjekts können dann die Metadaten ausgelesen werden (Bild 9).

Auf eine Betrachtung der Methoden des Reflection-API wird verzichtet und auf die SDK-Dokumentation verwiesen. Nur soviel: Man findet diese Methoden unter *System.Reflection*.

Attribute

Richtig interessant wird Reflection erst im Zusammenhang mit den sogenannten *Custom Attributes*. Das Common Type System bietet die Möglichkeit, jeden einzelnen Typ zur Entwicklungszeit mit eigenen Metadaten zu versehen. Diese Metadaten werden Attribute genannt und können zur Laufzeit ausgelesen werden. Neben der Verwendung vordefinierter Attribute besteht die Möglichkeit, eigene Attribute zu definieren. Attribute werden stets über Klassen implementiert, die sich von der Basisklasse *System.Attribute* ableiten. Schauen wir uns dazu ein Beispiel an. Listing 2 zeigt anhand von C#-Code, wie man einen generischen Mechanismus implementieren kann, der Klassen in einer Datenbank ablegt.

Mittels Reflection wird zunächst aus einer Klasse die dazugehörige Tabellendefinition erzeugt. Das erledigt die Methode *CreateSchema*. Dabei wird das Mapping zwischen Datenbanktypen und den Typ der einzelnen Klassenmembers über das Attribut *DBFieldType* realisiert.

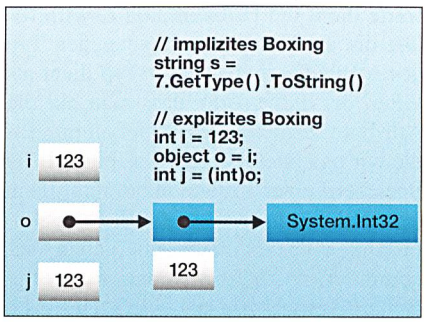

Bild 7 Boxing und Unboxing – aus der Sicht des Entwicklers sind auch ValueTypes Objekte

```
using System;
using System.Reflection;

namespace devcoach.de.BuilderDemo {

  [AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
  public class DBTypeAttribute : Attribute {
      public readonly string val;
      public DBTypeAttribute(string val) { this.val = val; }
  }

  public interface IBuilder {
      string CreateSchema(Type t);
      string Insert(object o);
  }

public class SQLBuilder : IBuilder {
      // Hilfsfunktion zum Auslesen des Attributwertes
      private string GetDBTypeAttribute(FieldInfo f) {
         DBTypeAttribute a = (DBTypeAttribute)Attribute.GetCustomAttribute(
             f, typeof(DBTypeAttribute)
             );
         if (null == a) return f.FieldType.ToString();
         return a.val;
      }

      public string CreateSchema(Type t) {
         FieldInfo[] fields = t.GetFields();
         int len = fields.Length;
         string sql = "create table (";
         FieldInfo f;
         for (int i = 0; i < len; i++) {
             f = fields[i];
             sql += f.Name + " " + GetDBTypeAttribute(f);
             if (i < len-1) sql += ",";
         }
         sql += ");";
         return sql;
      }

      public string Insert(object o) {
         Type t = o.GetType();
         FieldInfo[] fields = t.GetFields();
         int len = fields.Length;
         string sql = "insert into " + t.Name + " (";
         FieldInfo f;
         // Zuerst die Feldnamen ausgeben
         for (int i = 0; i < len; i++) {
             f = fields[i];
             sql += f.Name;
             if (i < len-1) sql += ",";
         }
         sql += ") values (";
         // Dann die Feldinhalte ausgeben
         for (int i = 0; i < len; i++) {
             f = fields[i];
             // Strings gehören in Quotes
             bool b = ((f.FieldType) == typeof(string));
             if (b) sql += "'";
             sql += f.GetValue(o).ToString();
             if (b) sql += "'";
             if (i < len-1) sql += ",";
         }
         sql += ");";
         return sql;
      }
  }
}
```

Listing 2 Generisches Speichern von Klassen – Reflection machts möglich

So wird der Code völlig unabhängig von der zugrunde liegenden Datenbank, und man legt den zu verwendenden Typ deklarativ fest. Zum Auslesen der Attributwerte dient die Hilfsfunktion *GetAttrValue*, die als Übergabeparameter den Typ des Attributs erhält. Dieser Typ dient als Filter, der dafür sorgt, dass man nur die Attribute zurückgeliefert bekommt, für die wir uns interessieren. Das eigentliche Speichern einer Instanz übernimmt dann die Methode *Insert*, welche die Klasseninstanz als Parameter übergeben bekommt. Um die Anwendung zu vereinfachen, verpackt man beide Funktionen in eine Klasse SQL-Builder (Listing 2 und 3).

Spinnt man diesen Faden weiter, könnte man beispielsweise auf die gleiche Art und Weise einen XML-Builder implementieren und eine Anwendung bauen, die zur Laufzeit entscheidet, wie eine Klasse abgespeichert werden soll. Entsprechende Klassen für das Arbeiten mit XML finden sich in der Klassenbibliothek unter *System.Xml* und *System.Xml.Xsl*. Über die Schnittstelle *IBuilder* wäre die Realisierung einer ClassFactory denkbar, die dann die eine geeignete Instanz des Builders zurückgibt.

Der Fantasie sind fast keine Grenzen gesetzt, und die Kombination von Reflection mit Attributen bietet viele, spannende Möglichkeiten. Ein Hinweis noch am Rand: Eine Attributklasse wird erst dann angelegt, wenn der erste Zugriff auf diese Klasse erfolgt, also zum Beispiel beim Aufruf der Methode *GetCustomAttributes*.

Assemblies und Versionierung

Der grösste Problembereich in der COM-Welt ist die Versionierung von Komponenten. Es ist unmöglich, unterschiedliche Versionen einer Komponente parallel zu installieren und zu nutzen. Eine nicht durchdachte Änderung kann weitreichende Folgen haben, da die Änderung sich auf alle Anwendungen auswirkt, die die Komponente benutzen. Dieses Problem ist unter der Bezeichnung DLL-Hölle wohlbekannt.

Um diesem Problem zu begegnen, hat Microsoft den Begriff *Assembly* eingeführt. Unter einem Assembly versteht man – vereinfacht gesagt – alle Komponenten, die eine Anwendung referenziert. Jedes Assembly verfügt über Metadaten, die die Abhängigkeiten der Komponenten beschreiben. Diese Metadaten werden *Manifest* genannt. Die einfachste Form ist ein *Private Assembly*. Dabei werden alle Komponenten in das Verzeichnis der Anwendung oder in ein Unterverzeichnis kopiert. Auf diese Art und Weise können verschiedene Anwendungen gleichzeitig mit verschiedenen Versionen einer Komponente arbeiten, ohne sich gegenseitig zu beeinflussen.

Sofern sich die Komponenten der Anwendung in einem oder mehreren Unterverzeichnissen befinden, kann man diese über eine Konfigurationsdatei im XML-Format spezifizieren. Der Name dieser Datei muss dem Namen der Anwendung entsprechen und die Endung CFG aufweisen (wenn eine Anwendung also beispielsweise *TEST.EXE* heisst, muss die Datei den Namen *TEST.CFG* haben). Ausserdem müssen sich Konfigurationsdatei und Anwendung im gleichen Verzeichnis befinden (Listing 4).

Wenn allerdings Komponenten von mehreren Anwendungen benutzt werden sollen, werden diese zu einem *Shared Assembly* zusammengefasst. Es ist dann möglich, das verschiedene Anwendungen gleichzeitig mit verschiedenen Versionen einer globalen Komponente arbeiten, ohne sich gegenseitig zu beeinflussen.

Ein Beispiel: Zwei Komponenten bilden ein Assembly und befinden sich im Verzeichnis V1.0.0.1. Dieses Assembly wird von drei unterschiedlichen Anwendungen benutzt. Nun wird eine neuere Version des Assemblies mit neuen Komponenten im Verzeichnis V1.1.0.1 installiert. Die Anwendungen können dann

```
using System;

namespace devcoach.de.BuilderDemo {
class Person {
    [DBType("integer")] public int age;
    [DBType("varchar(50)")] public string name;
    public Person (int age, string name) { this.age = age; this.name = name;
}
  }

  class App {
    static void Main(){
      IBuilder builder = new SQLBuilder();
      Console.WriteLine(builder.CreateSchema(typeof(Person)));
      Console.WriteLine(builder.Insert(new Person(32,"Michael Willers")));
    }
  }
}
```

Listing 3 Generisches Speichern zum Zweiten – ein kleines Testprogramm

```
<?xml version="1.0"?>
<configuration>
<runtime>
     . . .
     <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <probing privatePath="Stringer;Bin\Test" />
     </assemblyBinding>
     . . .
  </runtime>
</configuration>
```

Listing 4 Dateipfade werden über XML-Datei spezifiziert – Administratoren wirds freuen

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      . . .
      <dependentAssembly>
        <assemblyIdentity name="Revers" publicKeyToken="0038acc8beadf1e5"
/>
        <bindingRedirect oldVersion="2.0.0.0" newVersion="2.1.0.0" />
      </dependentAssembly>
      . . .
    </assemblyBinding>
  </runtime>
</configuration>
```

Listing 5 Anwendungen können gleichzeitig mit verschiedenen Versionen globaler Komponenten arbeiten

ebenfalls über eine Konfigurationsdatei bestimmen, mit welcher Assembly-Version sie arbeiten möchten (Listing 5).

Allerdings ist dabei zu beachten, dass dann natürlich wieder bestimmte Abhängigkeiten gelten und man sehr gut überlegen muss, wann und wie man eine Komponente verändert, wenn man nicht wieder geradewegs in die DLL-Hölle laufen will. Grundsätzlich bleibt aber festzuhalten, dass verschiedene Versionen ein und derselben Komponente mit Hilfe von Assemblies parallel installiert und ausgeführt werden können. Komponenteninformationen sind im Gegensatz zur COM-Welt nicht mehr nur systemweit gültig; sie können anwendungsspezifisch definiert werden.

Fazit

Die Common Language Runtime ermöglicht unabhängig von Programmiersprachen eine durchgängig objekt- und komponentenorientierte Programmierung. Es gibt ein einheitliches Integrationsmodell, und die Laufzeitumgebungen verschiedener Sprachen werden durch eine einzige Umgebung ersetzt. Compiler setzen auf dem Common Type System auf und erzeugen IL-Code. Die Integration erfolgt also im Gegensatz zu COM nicht auf binärer Ebene, sondern auf Codeebene – Sprachen sind per Definition interoperabel. Die Bedeutung, welche Sprache man zur Entwicklung von Anwendungen benutzt, rückt damit in den Hintergrund. Man arbeitet mit der Sprache, die einem am besten liegt. Oder anders formuliert: Sprachen werden gleichwertig und gewinnen an Bedeutung.

Die CLR und die dazugehörige Klassenbibiliothek abstrahieren die Entwicklung von einer konkreten Plattform. Weder COM noch das Win32-API sind zukünftig für den Entwickler sichtbar. Er programmiert mit der Klassenbibliothek, und die Compiler erzeugen prozessorunabhängigen IL-Code als Output.

Ein weiterer wichtiger Punkt: Das .NET Framework basiert nicht auf COM. Aber auch wenn COM nicht mehr benötigt wird, arbeitet das Framework nahtlos mit COM-Komponenten zusammen. Es ist von vorneherein auf Interoperabilität ausgelegt worden. Sie können COM-Komponenten aus .NET-Komponenten heraus benutzen und umgekehrt. Und nicht zuletzt wird die Installation von Anwendungen stark vereinfacht, da die Runtime nicht mehr auf der Registry aufsetzt. Verschiedene Versionen der gleichen Komponente können parallel eingesetzt

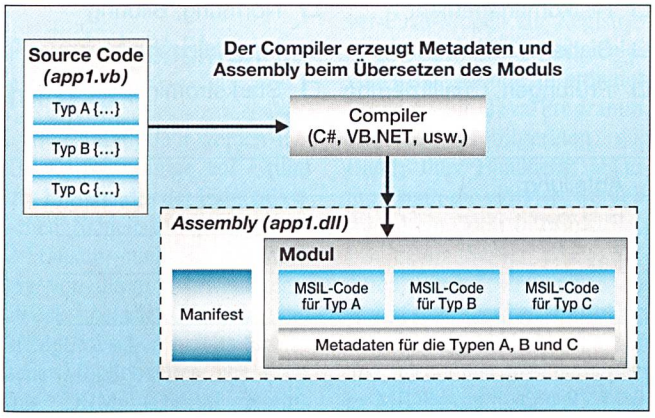

Bild 8 Typen werden immer durch Metadaten beschrieben, die vom Compiler erzeugt werden.

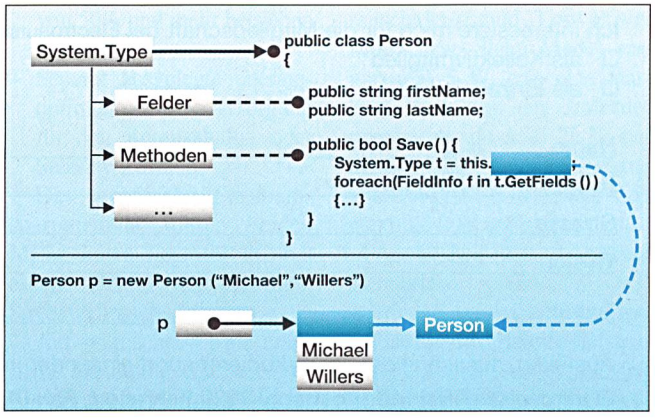

Bild 9 Der einzige Weg zu Informationen führt über das Typobjekt

werden. Die jedem Anwender und Entwickler wohlbekannte DLL-Hölle dürfte damit hoffentlich bald der Vergangenheit angehören.

Michael Willers, Initiator und Gründer des Entwicklerforums msdn TechTalk von Microsoft, war lange Jahre als Entwickler und Projektleiter tätig. Heute liegt der Schwerpunkt seiner Arbeit in der Vermittlung und Anwendung moderner Softwaretechnologien und -architekturen, insbesondere dem .NET Framework. Er ist Mitglied in verschiedenen Fachbeiräten und Lehrbeauftragter an Universitäten und Fachhochschulen in Deutschland. *Kontakt: michael.willers@devcoach.de, www.devcoach.de*

Le .NET Common Language Runtime – aperçu et initiation technique

Le .NET Framework est pour les développeurs l'élément le plus important de la nouvelle plate-forme .NET de Microsoft. La partie essentielle du Framework est le Common Language Runtime. Il vaut donc la peine de jeter un coup d'œil derrière les coulisses pour mieux saisir les concepts de cet environnement à base durée. L'article donne une vue d'ensemble et permet de se familiariser avec le sujet.

QFZ2002
SCHWEIZER QUALITÄTS-FACHZEITSCHRIFT AUSGEZEICHNET VON

Das Bulletin SEV/VSE gefällt mir und ich bestelle:
❑ 2 Gratis-Probeexemplare (unverbindlich)
❑ ein Jahresabonnement
❑ ab sofort ❑ ab .....

Fr. 205.– in der Schweiz

electrosuisse BULLETIN VSE AES

Publikationsorgan des SEV Verband für Elektro-, Energie- und Informationstechnik und des Verbandes Schweizerischer Elektrizitätsunternehmen VSE

Ich wünsche Unterlagen über
❑ Electrosuisse
❑ den Verband Schweiz. Elektrizitätsunternehmen (VSE)
❑ Inseratebedingungen

Ich interessiere mich für die Mitgliedschaft bei Electrosuisse
❑ als Kollektivmitglied
❑ als Einzelmitglied

Ich wünsche Unterlagen über folgende Tätigkeiten und Angebote der Electrosuisse:
❑ Total Security Management TSM®
❑ TSM Success Manuals
❑ Qualitätsmanagement ❑ Umweltmanagement
❑ Risikomanagement ❑ Normung, Bildung
❑ Sicherheitsberatung ❑ Innovationsberatung
❑ Prüfungen, Qualifizierung ❑ Starkstrominspektorat

Name
Firma Abteilung
Strasse PLZ/Ort
Telefon Fax
Datum Unterschrift

Ausfüllen, ausschneiden (oder kopieren) und einsenden an:
Electrosuisse, IBN MD, Postfach, 8320 Fehraltorf, Fax 01 956 11 22
Bestellungen auch über http://www.sev.ch